Jadavpur University

Department of Computer Science and Engineering

NETWORKS LAB ASSIGNMENT 2

BCSE UG-III

Student : Arjeesh Palai
Roll No. : 002310501086
Group : A3

Date : 08 / 09 / 2025

Problem Statement

Implement LLC flow control mechanisms over a simulated, potentially lossy and delayed channel. Compare Stop & Wait, Go-Back-N (GBN), and Selective Repeat (SR) ARQ in terms of throughput, retransmissions, RTT/RTO behavior, and correctness.

1 Design

Purpose. Simulate Data Link layer reliability using ARQ over an unreliable channel. Frames carry addresses, length, sequence number, payload, and CRC32 FCS. ACK/NAK frames are also protected by CRC.
Structure diagram.

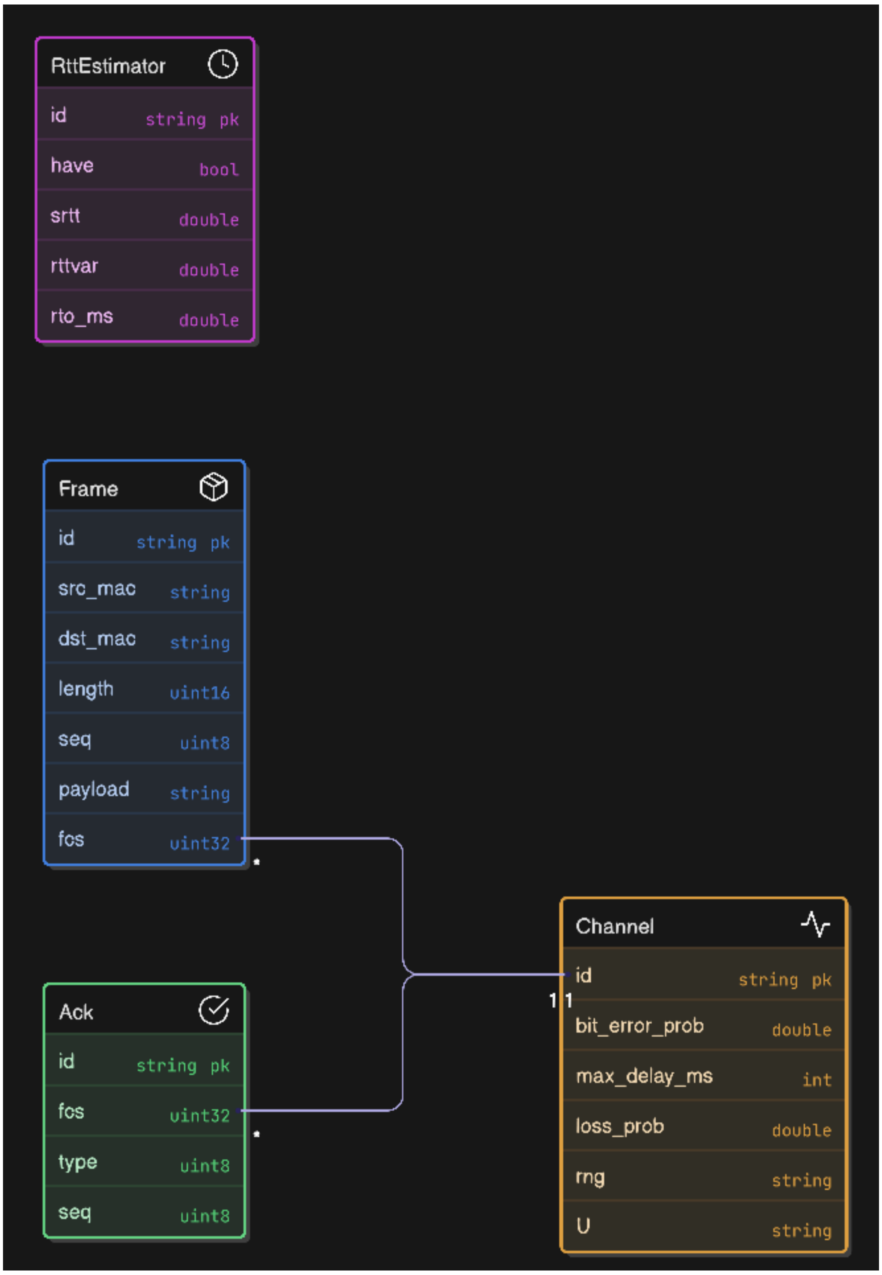

Figure 1: System structure overview

Input/Output

- **Input data**: `data.txt` generated by helper program (each line as payload; padded to minimum 46 bytes).
- **CLI parameters** (examples):
  - Stop&Wait sender: `p_err` (bit error probability), `max_delay` (ms)
  - Go-Back-N sender: `N` (sender window), `p_err`, `max_delay`
  - Selective Repeat sender: `N`, `p_err`, `max_delay`
- **Sockets**: TCP loopback port `8000`.

- **Output**: Console logs (sent/ACK/NAK/timeout), computed RTT/RTO, and successful delivery.

2 Implementation (Key Snippets Only)

2.1 Common utilities (`llc_common.h`)

`Channel (delay, loss, bit flips)`

```
static constexpr size_t MIN_PAYLOAD = 46;

struct Channel {
    double bit_error_prob = 0.0;
    int    max_delay_ms   = 0;
    double loss_prob      = 0.0;
    std::mt19937 rng{ std::random_device{}() };
    std::uniform_real_distribution<double> U{0.0, 1.0};

    void apply_delay() {
        if (max_delay_ms <= 0) return;
        int d = int(U(rng) * (max_delay_ms + 1));
        Sleep(static_cast<DWORD>(d));
    }
    bool maybe_drop() { return U(rng) < loss_prob; }

    void flip_bits(std::vector<uint8_t>& buf) {
        if (bit_error_prob <= 0.0) return;
        std::bernoulli_distribution B(bit_error_prob);
        for (auto& b : buf) {
            uint8_t m = 0;
            for (int i = 0; i < 8; ++i) if (B(rng)) m ^= (1u << i);
            b ^= m;
        }
    }
};
```

`Frame (serialize/parse + CRC32)`

```
struct Frame {
    uint8_t src[6]{}, dst[6]{};
    uint16_t length{0};
    uint8_t seq{0};
    std::vector<uint8_t> payload;
    uint32_t fcs{0};
```

```

    std::vector<uint8_t> serialize_with_crc() {
        std::vector<uint8_t> body;
        body.insert(body.end(), src, src + 6);
        body.insert(body.end(), dst, dst + 6);
        uint16_t be_len = htons(length);
        body.push_back(uint8_t(be_len >> 8));
        body.push_back(uint8_t(be_len & 0xFF));
        body.push_back(seq);
        body.insert(body.end(), payload.begin(), payload.end());
        if (body.size() < (15 + MIN_PAYLOAD))
            body.insert(body.end(), (15 + MIN_PAYLOAD) - body.size(), uint8_t(' '));
        uint32_t c = crc32(body.data(), body.size());
        fcs = c;
        body.push_back(uint8_t((c >> 24) & 0xFF));
        body.push_back(uint8_t((c >> 16) & 0xFF));
        body.push_back(uint8_t((c >> 8) & 0xFF));
        body.push_back(uint8_t(c & 0xFF));
        return body;
    }

    static bool parse(const std::vector<uint8_t>& buf, Frame& out) {
        if (buf.size() < 15 + MIN_PAYLOAD + 4) return false;
        std::copy(buf.begin(), buf.begin() + 6, out.src);
        std::copy(buf.begin() + 6, buf.begin() + 12, out.dst);
        uint16_t be_len = (uint16_t(buf[12]) << 8) | uint16_t(buf[13]);
        out.length = ntohs(be_len);
        out.seq = buf[14];
        size_t pay = std::max<size_t>(MIN_PAYLOAD, out.length);
        if (buf.size() < 15 + pay + 4) return false;
        out.payload.assign(buf.begin() + 15, buf.begin() + 15 + pay);
        out.fcs = (uint32_t(buf[15+pay]) << 24) | (uint32_t(buf[16+pay]) << 16)
                | (uint32_t(buf[17+pay]) << 8) | uint32_t(buf[18+pay]);
        return true;
    }

    static bool verify_crc(const std::vector<uint8_t>& buf) {
        if (buf.size() < 4) return false;
        uint32_t got = (uint32_t(buf[buf.size()-4]) << 24) |
                       (uint32_t(buf[buf.size()-3]) << 16) |
```

```
                        (uint32_t(buf[buf.size()-2]) << 8)  |
                         uint32_t(buf[buf.size()-1]);
        uint32_t calc = crc32(buf.data(), buf.size() - 4);
        return got == calc;
    }
};
```

ACK/NAK with CRC

```
enum : uint8_t { ACK = 0x06, NAK = 0x15 };

struct Ack {
    uint8_t type{ACK};
    uint8_t seq{0};
    uint32_t fcs{0};

    std::vector<uint8_t> serialize() {
        std::vector<uint8_t> b{type, seq};
        uint32_t c = crc32(b.data(), b.size());
        fcs = c;
        b.push_back(uint8_t((c >> 24) & 0xFF));
        b.push_back(uint8_t((c >> 16) & 0xFF));
        b.push_back(uint8_t((c >> 8) & 0xFF));
        b.push_back(uint8_t(c & 0xFF));
        return b;
    }

    static bool parse(const uint8_t* buf, size_t len, Ack& out) {
        if (len < 6) return false;
        out.type = buf[0];
        out.seq  = buf[1];
        uint32_t got = (uint32_t(buf[2]) << 24) | (uint32_t(buf[3])
           << 16)
                     | (uint32_t(buf[4]) << 8)  |  uint32_t(buf[5]);
        std::vector<uint8_t> b{out.type, out.seq};
        uint32_t calc = crc32(b.data(), b.size());
        if (got != calc) return false;
        out.fcs = got; return true;
    }
};
```

RTO Estimator (Jacobson/Karels)

```
struct RttEstimator {
```

```
    bool have=false; double srtt=0.0, rttvar=0.0; double rto_ms
       =1000.0;
    void observe(double sample_ms) {
        if (!have) { srtt=sample_ms; rttvar=sample_ms/2.0; have=true
           ; }
        else {
            const double alpha=1.0/8.0, beta=1.0/4.0;
            rttvar = (1.0 - beta)*rttvar + beta*std::abs(srtt -
               sample_ms);
            srtt   = (1.0 - alpha)*srtt   + alpha*sample_ms;
        }
        rto_ms = clampd(srtt + 4.0*rttvar, 200.0, 4000.0);
    }
};
```

2.2 Stop & Wait (core loops)

Sender: send → wait-ACK with adaptive RTO

```
auto wire = f.serialize_with_crc();
bool acked = false;
while (!acked) {
    chan.apply_delay();
    auto tx = wire;
    chan.flip_bits(tx);
    if (!chan.maybe_drop()) send_all(conn, tx.data(), tx.size());
    std::cout << "[SENDER] Sent frame seq=" << int(seq) << "\n";

    auto t0 = std::chrono::steady_clock::now();
    uint8_t ackbuf[6];
    if (recv_exact(conn, ackbuf, sizeof(ackbuf), int(rtt.rto_ms))) {
        Ack a{};
        if (Ack::parse(ackbuf, sizeof(ackbuf), a) && a.type==ACK &&
           a.seq==seq) {
            auto t1 = std::chrono::steady_clock::now();
            double ms = std::chrono::duration<double,std::milli>(t1
               - t0).count();
            rtt.observe(ms);
            std::cout << "[SENDER] ACK " << int(a.seq)
                      << " (RTT=" << ms << "ms, RTO=" << rtt.rto_ms
                         << "ms)\n";
            acked = true; seq = uint8_t(seq + 1);
        } else {
            std::cout << "[SENDER] Bad ACK/NAK; retransmitting\n";
```

```
        }
    } else {
        std::cout << "[SENDER] Timeout; retransmitting seq=" << int(
           seq)
                  << " (RTO=" << rtt.rto_ms << "ms)\n";
    }
}
```

Receiver: CRC+SEQ check; ACK only if in-order

```
bool ok_crc = Frame::verify_crc(buf);
Frame f{}; bool parsed = Frame::parse(buf, f);
if (!parsed) { /* drop */ }
std::cout << "[RECV] Frame seq=" << int(f.seq)
          << " CRC=" << (ok_crc ? "OK" : "BAD") << "\n";

if (ok_crc && f.seq == expected) {
    expected = uint8_t(expected + 1);
    Ack a{ACK, f.seq};
    auto wire = a.serialize();
    chan.apply_delay(); chan.flip_bits(wire);
    if (!chan.maybe_drop()) send_all(s, wire.data(), wire.size());
    std::cout << "[RECV] ACK sent for " << int(f.seq) << "\n";
} else {
    std::cout << "[RECV] Discarded (crc/seq mismatch). No ACK.\n";
}
```

2.3 Go-Back-N (key logic)

Sender: pipeline, cumulative ACKs, timeout ⇒ resend window

```
uint8_t base=0, nextseq=0; int N=4;
std::map<uint8_t,std::vector<uint8_t>> frame_cache;

auto in_window = [&](uint8_t s){
  int diff = int(uint8_t(s - base));
  return 0 <= diff && diff < N;
};

auto send_frame = [&](uint8_t seq, const std::vector<uint8_t>&
   payload){
  Frame f; /* fill header + payload, set f.seq=seq */
  auto w_clean = f.serialize_with_crc();
  frame_cache[seq] = w_clean;
```

```
  auto w = w_clean; chan.apply_delay(); chan.flip_bits(w);
  if (!chan.maybe_drop()) send_all(conn, w.data(), w.size());
  std::cout << "[GBN SENDER] Sent seq=" << int(seq) << "\n";
};

while (base != nextseq || more_data()) {
  while (in_window(nextseq) && more_data())
      { send_frame(nextseq, next_payload()); nextseq = uint8_t(nextseq + 1); }

  uint8_t ackbuf[6];
  if (recv_exact(conn, ackbuf, sizeof(ackbuf), int(rtt.rto_ms))) {
      Ack a{}; if (Ack::parse(ackbuf, sizeof(ackbuf), a) && a.type==ACK) {
          if (int(uint8_t(a.seq - base)) > 0) {
              base = a.seq; prune_cache_before(base);
          }
      }
  } else {
      std::cout << "[GBN SENDER] TIMEOUT, resending ["<<int(base)<<","<<int(nextseq)<<")\n";
      for (uint8_t s = base; s != nextseq; s = uint8_t(s + 1)) {
          auto it = frame_cache.find(s);
          if (it != frame_cache.end()) {
              auto w2 = it->second; chan.apply_delay(); chan.flip_bits(w2);
              if (!chan.maybe_drop()) send_all(conn, w2.data(), w2.size());
          }
      }
  }
}
```

Receiver: expected seq, CRC check, send cumulative ACK

```
uint8_t expected = 0;
bool ok = Frame::verify_crc(buf);
Frame f{}; Frame::parse(buf, f);
if (ok && f.seq == expected) expected = uint8_t(expected + 1);
Ack a{ACK, expected};             % cumulative ACK for next expected
auto w = a.serialize();
chan.apply_delay(); chan.flip_bits(w);
if (!chan.maybe_drop()) send_all(s, w.data(), w.size());
```

2.4 Selective Repeat (key logic)

`Sender: per-slot timers, selective retransmit on NAK/timeout`

```
struct Slot {
  bool in_use=false, acked=false;
  std::vector<uint8_t> wire;
  std::chrono::steady_clock::time_point deadline;
};
uint8_t base=0, nextseq=0; int N=6;
std::map<uint8_t,Slot> window;

auto in_window = [&](uint8_t s){ return int(uint8_t(s - base)) >= 0
                                   && int(uint8_t(s - base)) < N; };

auto send_or_resend = [&](uint8_t seq){
  auto &slot = window[seq]; auto w = slot.wire;
  chan.apply_delay(); chan.flip_bits(w);
  if (!chan.maybe_drop()) send_all(conn, w.data(), w.size());
  slot.deadline = std::chrono::steady_clock::now()
                + std::chrono::milliseconds(int(rtt.rto_ms));
};

while (!done()) {
  % push new frames
  while (in_window(nextseq) && more_data()) {
    Frame f; /* fill header+payload; f.seq=nextseq */
    window[f.seq] = Slot{true,false,f.serialize_with_crc(),{}};
    send_or_resend(f.seq);
    nextseq = uint8_t(nextseq + 1);
  }

  % handle ACK/NAK
  if (recv_ack_or_nak(a)) {
    if (a.type==ACK && window.count(a.seq)) {
      window[a.seq].acked = true;
      while (window.count(base) && window[base].acked) { window.erase(base); base=uint8_t(base+1); }
    } else if (a.type==NAK && window.count(a.seq)) send_or_resend(a.seq);
  }

  % handle timeouts
  for (auto &kv : window)
```

```
    if (!kv.second.acked && now() >= kv.second.deadline)
       send_or_resend(kv.first);
}
```

```
Receiver: buffer out-of-order, ACK each valid, NAK on CRC error
```

```
uint8_t base=0; int N=6; std::map<uint8_t,Frame> buffer;

bool ok = Frame::verify_crc(buf);
Frame f{}; if (!Frame::parse(buf, f)) return;

if (!ok) {
  Ack n{NAK, base}; auto w = n.serialize();
  chan.apply_delay(); chan.flip_bits(w);
  if (!chan.maybe_drop()) send_all(s, w.data(), w.size());
} else {
  int diff = int(uint8_t(f.seq - base));
  if (diff < 0) {                          // valid duplicate
    Ack a{ACK, f.seq}; auto w=a.serialize();
    chan.apply_delay(); chan.flip_bits(w);
    if (!chan.maybe_drop()) send_all(s, w.data(), w.size());
  } else if (diff < N) {                   // in-window
    buffer[f.seq] = f;
    Ack a{ACK, f.seq}; auto w=a.serialize();
    chan.apply_delay(); chan.flip_bits(w);
    if (!chan.maybe_drop()) send_all(s, w.data(), w.size());
    while (buffer.count(base)) { buffer.erase(base); base = uint8_t(
       base + 1); }
  }
}
```

2.5 Data generator (payload maker)

```
make_data.cpp: generate data.txt
```

```
int main() {
  std::mt19937 rng(12345);
  std::uniform_int_distribution<int> lenDist(10,120), byteDist
     (0,255);
  std::ofstream out("data.txt", std::ios::binary);
  for (int i=1;i<=10;++i) {
    int L = lenDist(rng);
    std::vector<unsigned char> buf; buf.reserve(L);
    for (int j=0;j<L;++j) {
```

```
      unsigned char b; do { b=byteDist(rng); } while (b==0x0A || b
         ==0x0D);
      buf.push_back(b);
    }
    out.write(reinterpret_cast<const char*>(buf.data()), buf.size())
       ;
    out.put('\n');
  }
  return 0;
}
```

3 Test Cases (Commands Used)

Stop-and-Wait

```
# TC1     Baseline (no error/loss)
Terminal A: stopwait_sender.exe 0 0
Terminal B: stopwait_receiver.exe 0 0

# TC2     Delay only
Terminal A: stopwait_sender.exe 0 120
Terminal B: stopwait_receiver.exe 0 120

# TC3     Mixed (moderate errors + delay)
Terminal A: stopwait_sender.exe 0.0005 100
Terminal B: stopwait_receiver.exe 0.0005 100
```

Go-Back-N

```
# TC1     Baseline, N=4
Terminal A: gobackn_sender.exe 4 0 0
Terminal B: gobackn_receiver.exe 0 0

# TC2     Delay only, N=4
Terminal A: gobackn_sender.exe 4 0 150
Terminal B: gobackn_receiver.exe 0 150

# TC3     Mixed, N=8
Terminal A: gobackn_sender.exe 8 0.0005 100
Terminal B: gobackn_receiver.exe 0.0005 100
```

Selective Repeat

```
# TC1     Baseline, N=4
Terminal A: sr_sender.exe 4 0 0
```

```
Terminal B: sr_receiver.exe 4 0 0

# TC2     Delay only, N=5
Terminal A: sr_sender.exe 5 0 120
Terminal B: sr_receiver.exe 5 0 120

# TC3     Mixed, N=6
Terminal A: sr_sender.exe 6 0.0005 100
Terminal B: sr_receiver.exe 6 0.0005 100
```

4 Results (Observed Behaviour)

Stop-and-Wait

TC1 (0,0): Receiver logs only `CRC=OK`. Sender prints `ACK <seq>` for each frame; *no timeouts*.
TC2 (0,120): Receiver `CRC=OK`. Sender shows occasional `Timeout; retransmitting seq=<n>`, then progresses once ACK arrives (idle waiting visible).
TC3 (0.0005,100): Receiver mixes `CRC=OK`/`CRC=BAD`. Sender retries same `seq` until ACKed; steady but slower progress.

Go-Back-N

TC1 (N=4, 0,0): Receiver starts `seq=0 CRC=OK expected=0`, responds with `cumulative ACK=1`, etc. No `CRC=BAD`, no timeouts.
TC2 (N=4, 0,150): Sender prints periodic `TIMEOUT, resending [b,e)`. Receiver mostly `CRC=OK`; out-of-order or corrupted frames discarded; cumulative ACK held until the gap closes.
TC3 (N=8, 0.0005,100): Receiver intermittently `CRC=BAD`, discards out-of-order until missing seq is received, then `Sent cumulative ACK=<k>`. Sender times out and resends current window; progress in bursts.
Typical receiver log:

```
[GBN RECV] seq=5 CRC=BAD expected=5 -> discard
[GBN RECV] seq=7 CRC=OK expected=5 -> discard
[GBN RECV] seq=5 CRC=OK expected=5
[GBN RECV] Sent cumulative ACK=6
```

Selective Repeat

TC1 (N=4, 0,0): Receiver ACKs each frame; no NAKs/timeouts.
TC2 (N=5, 0,120): Sender shows `Timeout seq=<n> -> retransmit`. Receiver accepts in-window out-of-order, advances `base` as gaps fill.
TC3 (N=6, 0.0005,100): Receiver: `CRC=BAD` ⇒ `NAK <base>`; buffers valid out-of-order; re-ACKs valid duplicates (`seq < base`); `base` increments when gap fills. Sender performs

targeted retransmissions; progress resumes after each NAK/timeout.
Typical receiver log:
```
[SR RECV] seq=1 CRC=BAD base=1 -> NAK 1
[SR RECV] seq=2 CRC=OK base=1 -> ACK 2
[SR RECV] seq=1 CRC=OK base=1 -> ACK 1
```

5 Discussion

- **S&W**: Deterministic progress in TC1; TC2 shows idle time dominated by propagation/queuing delay; TC3 shows reliability via repeat-until-ACK with reduced throughput.
- **GBN**: TC1 confirms pipeline gains without penalties. TC2 highlights bursty window timeouts due to delayed ACK arrivals; TC3 shows classic burst retransmissions and cumulative ACK jumps once the missing seq arrives.
- **SR**: TC1 clean; TC2 demonstrates selective timeout handling per-slot; TC3 validates fine-grained recovery (NAKs + individual ACKs), highest efficiency under errors at the cost of buffering/state.
- **Across schemes**: Increasing delay magnifies RTO sensitivity; errors penalize GBN most (window-wide retransmits) and SR least (targeted retries).

6 Diagrams

Flow-control diagrams

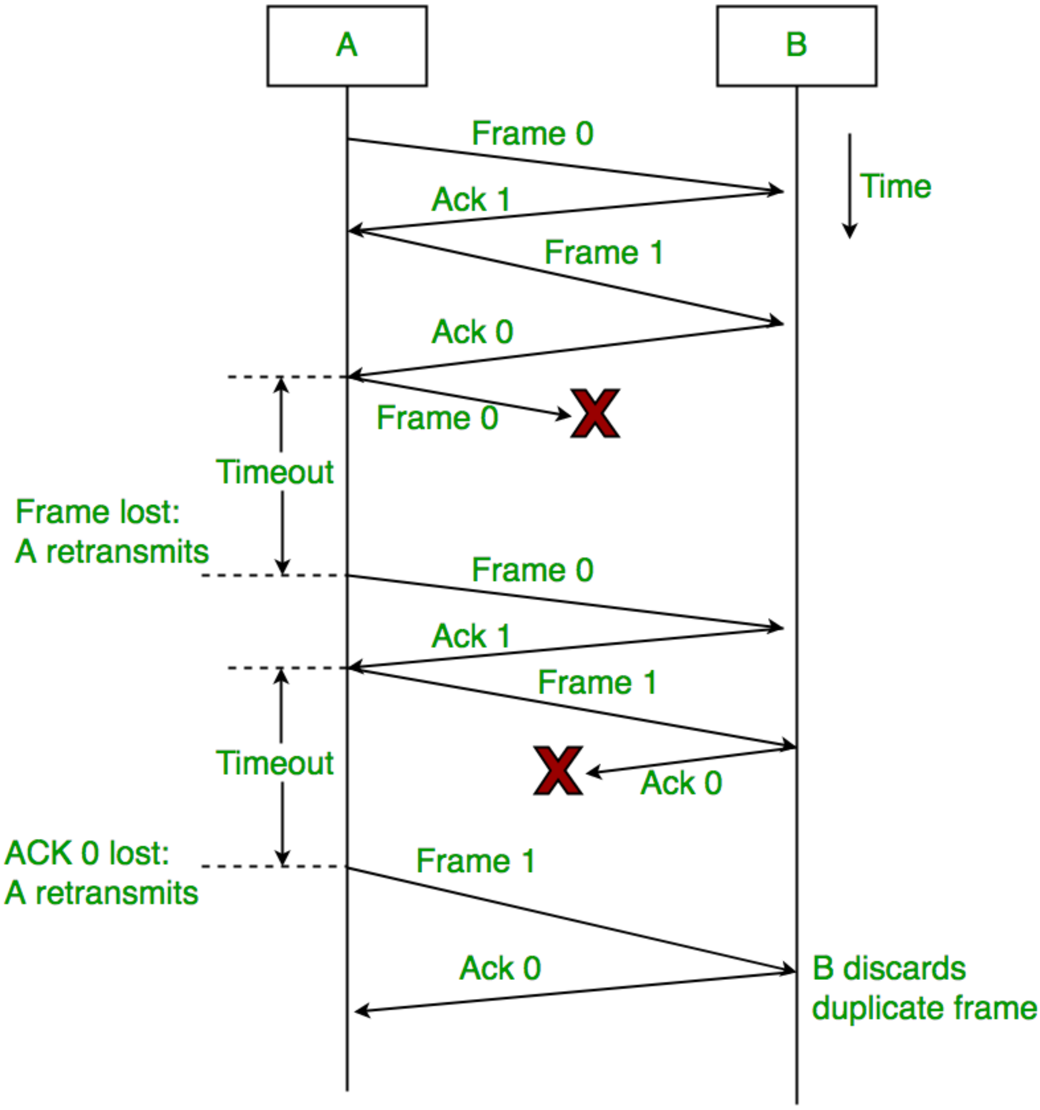

Figure 2: Stop & Wait

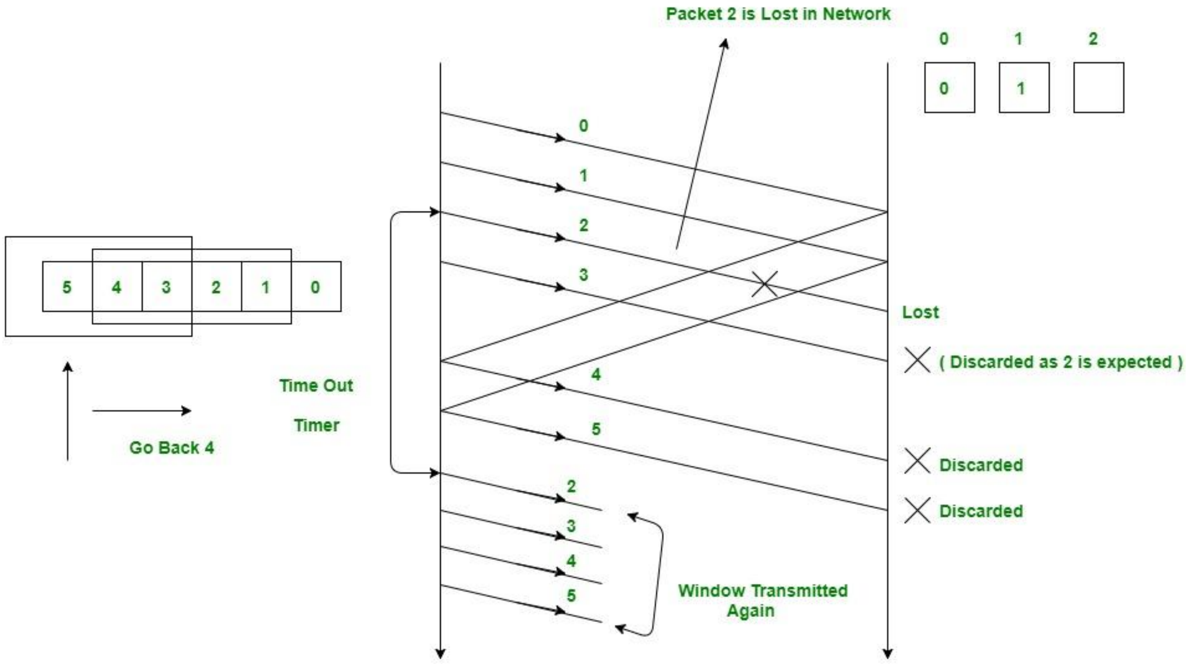

Figure 3: Go-Back-N ARQ

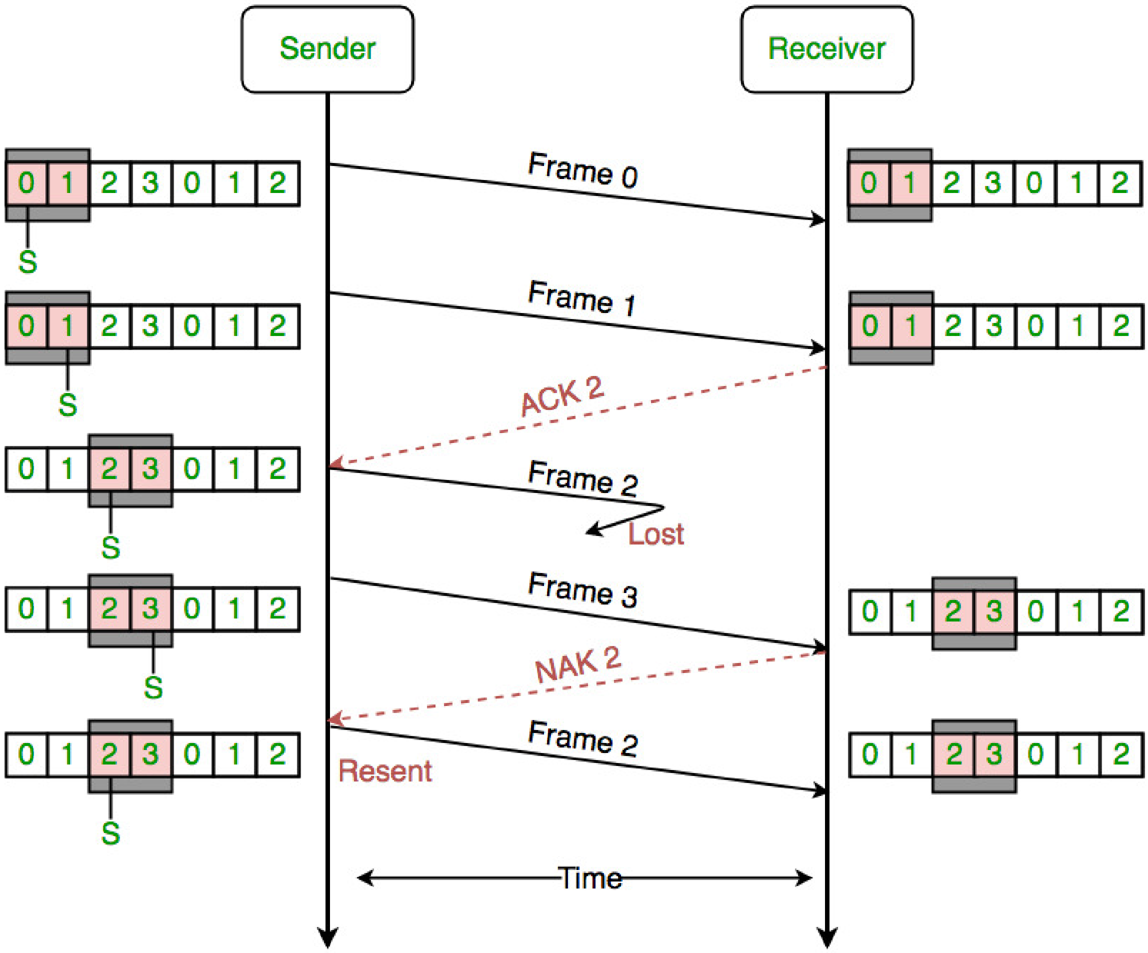

Figure 4: Selective Repeat ARQ